Evaluating the Tracing of Recursion in the Substitution Notional Machine

Preston Tunnell Wilson
Brown University, USA
ptwilson@brown.edu

Kathi Fisler
Brown University, USA
kfisler@cs.brown.edu

Shriram Krishnamurthi
Brown University, USA
sk@cs.brown.edu

ABSTRACT

We evaluate a notional machine for recursion based on algebraic substitution. To do this, we decompose recursion into a progression of function call patterns, parameter name reuse, and data structure complexity. At each stage, we test students' ability to trace programs using substitution. We evaluate the correctness of their traces along multiple dimensions, finding that students generally do well, and also observe shortcuts and identify misconceptions. For comparison, we also have students trace two problems using a traditional, imperative notional machine. Even though the substitution model is unwieldy to use with compound data, students still perform better with it than with the traditional notional machine.

ACM Reference Format:
Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. 2018. Evaluating the Tracing of Recursion in the Substitution Notional Machine. In *SIGCSE '18: SIGCSE '18: The 49th ACM Technical Symposium on Computer Science Education, February 21–24, 2018, Baltimore , MD, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3159450.3159479

1 INTRODUCTION

Many authors (e.g., [6, 7]) have discussed student difficulties with recursion. This paper focuses on students' ability to *trace* recursive programs using a notional machine [2]. Most computing pedagogy uses an imperative, stack-based notional machine. We instead study a rarely-used model, based on *algebraic substitution* (section 3), from *How to Design Programs* [4] (HTDP). Though this model has been realized as a tool [1], it has not been evaluated before.

When tracing recursion, a student must track both where functions return and what different values are bound to the same variable name. Thus, instead of viewing recursion as an atomic activity, we observe that it can follow a learning progression: a student must understand what happens when (a) one function calls another function and control returns; (b) variable names are reused across functions; and (c) putting these together, when a function calls and returns from itself (in which case, variable names are necessarily reused). We use this progression to structure our study, observing which skill causes problems. We also have students use tree-shaped data, which are not covered in many prior recursion studies.

SIGCSE '18, February 21–24, 2018, Baltimore , MD, USA

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159479

2 RELATED WORK

There is an extensive body of literature on recursion in computing education. We focus just on the most relevant papers. Though Settle [12] has a bibliography on recursion, it does not discuss the substitution model at all, which is the focus of this paper.

Lewis [8] conducts talk-alouds on two recursive problems from the 1988 AP CS A exam. She presents students' traces and highlights four mental models of how substitution in recursive problems works. We include her questions in our study. We do not observe the same four mental models as she does, but this may be due to differences in format (her talk-alouds versus our written quizzes). Unlike Lewis, we discuss difficulties weaker students had with tracing rather than focusing solely on successful traces.

McCauley et al. [9] test students' ability to comprehend recursive and iterative programs. There are many differences between our efforts. They do not ask for traces; they include compound data but only linked lists, not trees; their programs contain imperative updates; and in general, we do not compare recursion and iteration (which is anyway not straightforward on trees).

Tessler et al. [13] use a game to contextualize recursive operations, lecturing students on recursion either before or after the game. They compare students' traces and solutions between these two conditions. All of our students are in the same condition, but they use two different notional machines during the study.

Kurland et al. [7] ask students to think aloud about what recursive graphical Logo programs would do, then draw out what they think the program would do. This is more of a comprehension task than one on tracing—these drawings are only loosely tracings.

Wiedenbeck [15] studies how students learn iteration and recursion and how learning one impacts the other. She focuses on using examples as the main pedagogic tool. Our focus is instead on tracing, the substitution notional machine, and includes tree recursion, which is not easily done with iteration.

Nelson et al. [10] use a combination of notional machines and tracing to teach programming and compare their tutorial to Codeacademy. However, their notional machine is much lower level than ours, with explicit stack manipulation, a program counter, and setting values in a namespace, all of which are absent or attenuated in substitution. Also, rather than analyzing students' traces, they provide traces to students through example programs.

Kahney [6] experiments on how novices' understanding of recursion differs from that of experts. His focus is on comprehension rather than on tracing, giving students multiple purported solutions and asking which ones would work. We also do not ask students for their formal reasoning, nor did we compare against experts.

Sanders et al. [11] observe students' mental models on recursion and make changes to their pedagogy to improve their models. We propose an alternative pedagogy based on the substitution model.

```
fun f(x):
  (2 * x) + 15
end
```

```
1 f(3 + 2)
2 f(5)
3 (2 * 5) + 15
4 10 + 15
5 25
```

Figure 1: Sample function definition and trace through it

```
fun f(x):
  3 + g(5)
end

fun g(y):
  x + y
end
```

```
1 f(2)
2 3 + g(5)
3 3 + (x + 5)
```

Figure 2: How substitution prevents dynamic scope

3 BACKGROUND: SUBSTITUTION MODEL

The substitution model is familiar from algebra classes: when a function is applied to arguments (the "actual parameters"), first the arguments are evaluated, then all instances of the function's formal parameters are replaced with their argument values. The resulting program is then evaluated using these concrete values.

Figure 1 shows an example of substitution in Pyret [pyret.org], a Python-inspired, student-friendly programming language used predominantly in the studies in this paper. The left shows the definition of a function `f`, and the right a trace using substitution. Line 1 shows the initial call. In line 2, the actual parameter is reduced to a value, 5. Line 3 is the crucial *substitution* step: the body of `f` is rewritten with all instances of the formal parameter, `x`, substituted with the value 5. In lines 4 and 5, evaluation proceeds as expected using the rules of arithmetic.

When a student traces through a function call using substitution, they write a new line each time a new expression is evaluated. In the process they copy the pending computations, or accumulated *context*, from the previous line. Thus, in the transition from lines 3 to 4, while the expression `2 * 5` evaluates to `10`, the context of `+ 15` is copied. Note that this copied context corresponds to a stack, but without having to introduce it as a new concept. This notional machine is explained in complete detail in HTDP.

For programs without mutation, substitution is a viable notional machine that avoids the need for stacks, stores, and other representations of memory. We hypothesize that the substitution model offers students a mechanical, consistent, and simple way of tracing recursion. An expert can see that substitution encodes the "copies" model and other concepts explored in earlier literature [6, 11].

One consequence of substitution is that it is easy to see whether or not a variable is bound. Consider fig. 2. On the left, the function `g` has an unbound variable (`x`). This is an error in all languages that obey static scope (which is most modern languages), though prior studies have shown that both novices [5] and professionals [14] can be confused by this, thinking the `x` of `f` is still "visible" in `g`. On the right, we show a trace with substitution. Because, on function application, substitution replaces all parameters with their actual values, *any remaining variables must not be bound*. In contrast, in a traditional notional machine with an explicit stack and store, if the student forgets to "drop" `x` on exit from `f`, they would assume `x` were bound.

However, the substitution model, as we have described it, requires a significant extension to model mutation [3]. For the Python 3 program on the left, consider the trace (right) of the call `g(10)`:

```
1 def g(y):
2   y = y + 2
3   z = y + 1
4   return z
```

```
1 y = 10 + 2
2 z = 10 + 1
3 return 10 + 1
```

The trace result is clearly wrong. The error is because we assumed `y` would not change, but the assignment (`y = `) changes it. Thus, students must deal with time, looking up the *current* values of variables. Therefore, many textbooks and authors explicitly or implicitly use a *mutable* model, one where names are held in dictionaries that can be updated as the program progresses. Note that in this model, students must remember to explicitly drop variables as they exit scopes, to avoid the dynamic scope problem described above.

4 STUDY DESIGN

4.1 A Learning Progression for Recursion

We break down the tracing of a recursive program into a progression of separate concerns that a student must understand:

- function calls (dealing with parameters),
- function returns (dealing with the context),
- parameter name reuse across different functions,
- calling the same function more than once nonrecursively, and
- a function calling itself (which combines the above two).

We conjecture that teaching these concerns one-by-one might make recursion easier to follow. (Note that the first four concerns have nothing to do with recursion per se, and hence are independently useful.) Our formal research questions assess students' ability to trace programs through the lens of this progression (as reflected concretely in the study problems (section 4.2)), and how students use the substitution model within each stage:

RQ1: Where in the progression do students struggle?
RQ2: As students use the model, what misconceptions appear?
RQ3: What aspects of the model do students avoid with shortcuts?

4.2 Evaluation Programs

The above progression translates into a set of programs that students are asked to trace. Representatives are shown in fig. 3. In (a) we see just a function that uses its parameter. In (b) we have `f` calling `g`, with both having the same parameter name. (Observe that in substitution there should not be any confusion between the two `x`'s, because the first `x` has "disappeared"—due to having been substituted—at the point of calling `g`.) In (c) we use a function twice, to see whether this new concept creates any problems. In (d) we show recursion over recursive data (rather than over numbers). This function computes the product of all the numbers in the list. In (e) we introduce a tree data structure and write a recursive function that computes the height of the tree. Observe that in both (d) and (e)

```
# a: Simple function call
fun f(a):
  a + (5 * a)
end
```

```
# b: Parameter name reuse
fun f(x):
  x + g(x * 2)
end

fun g(x):
  x - 3
end
```

```
# c: Function call reuse
fun f(x):
  g(g(x - 5)) - 2
end

fun g(y):
  3 * y
end
```

```
# d: Recursion over list
fun h(lst):
  cases(List) lst:
    | empty => 1
    | link(f, r) =>
      f * h(r)
  end
end
```

```
# e: Recursion over tree
data BinTree:
  | leaf()
  | node(value, left, right)
end

fun f(t):
  cases(BinTree) t:
    | leaf() => 1
    | node(v, l, r) =>
      num-max(f(l), f(r)) + 1
  end
end
```

Figure 3: Sample programs

the recursion is non-trivial, i.e., it is not tail-recursion (equivalent to a loop).

In addition to these types of problems, we use three more. Two are from an AP CS test and were used in a prior paper [8]. The third is a purely numeric recursion implementing the Collatz function, a.k.a., "$3n + 1$" (halt at $n = 1$; for even positive n, divide by two and recur; for odd positive n, recur on $3n + 1$).

4.3 Logistics: From Problems to Quizzes

To evaluate student understanding, we converted these problems into quizzes/homeworks to administer in a class.

Class Context and Participants. The quizzes were administered in a summer course (equivalent in content and credit to a regular semester-long course) at a selective, private US university. The course, an introduction to programming with no prerequisites, covered basic expressions, functions, recursion over lists and trees, variables, mutation (of both variables and fields within objects), and conventional loops. The 19 students ranged from high-school students to adult learners embarking on a professional master's degree in data science. One of the authors taught the course.

Quiz Logistics. All quizzes were given on paper. The first three were given in class; students had 15–20 minutes to complete each one. The last two were unlimited-time take-home exercises. The students knew that the quizzes were part of an ongoing course diagnostic, and had little to no impact on overall course grades. All quizzes were reviewed in class immediately after submission, so students received feedback on their work. The quizzes were given roughly once every 5-7 calendar days.

We administered a total of five quizzes, summarized below. The first four were in Pyret, while the fifth was in Python 3 (matching the course structure, which began in Pyret and ended in Python).[1]

(1) Substitution into functions performing arithmetic on numbers (fig. 3(a)) and concatenation on strings; calling two functions (like fig. 3(b), but *without* parameter name reuse—the functions had different parameter names); and a dynamic scope question (akin to fig. 2). For this quiz only, students were given the result to a function call and asked to explain (through tracing) how that answer ensued, to focus on tracing rather than on correctness. In subsequent quizzes, students also had to determine the answer. The dynamic scope question was phrased as "Explain what happens if we evaluate `f(2)`: what outcome do we get and why?"
(2) Shared parameter names between functions (fig. 3(b)); multiple calls to the same function (fig. 3(c)); another dynamic scope question; and the Collatz function as a pre-test, because recursion had not yet been taught in class. Questions were phrased as "Show how the program `f(7)` evaluates to an answer."
(3) Linear structural [4] recursion (fig. 3(d)). A sample question is "Show how `h([list: 5, 0, -2])` evaluates to an answer."
(4) Structural recursion over trees (fig. 3(e)); a tweak on the Fibonacci function; and the two AP CS test questions [8]. A sample question is "What does calling `g(4)` produce? List the sequence of function calls that get made, with the actual arguments to `g`, showing how the program arrives at its answer."
(5) Recursion over trees and lists of tuples, done in Python.

By the last quiz the class had transitioned to Python, where they were introduced to mutable variables as well as tuples. They were also taught a dictionary-based traditional notional machine. The "stack" was represented by a process of adding to and removing from the dictionary. Thus, they were effectively repeating the previous two quizzes, but in a different language and, more importantly, using a different notional machine.

In the quiz instructions, students were initially asked to "write out the computations showing how [the question] evaluates." We expected students to write out every step, as illustrated in section 3. As the programs became more complex (involving conditionals and data structures), we relaxed the steps that the students had to show:

[1] The details are on the Web at cs.brown.edu/research/plt/dl/sigcse2018-recursion/.

the revised instructions added "You only need to show steps with function calls or the results of computations on numbers/strings/etc. Don't write out `if` or `cases` steps."

5 RESULTS

Given our goal to study whether students could effectively trace recursive programs, our analysis examined the traces that students provided as quiz answers. We defined multiple ways to compare their traces against a ground-truth trace written by an expert. (The substitution model has a long history, so there is widespread agreement on what the ground-truth trace ought to be.) In performing these comparisons, we noted shortcuts taken by students, as well as errors.

We defined the following measures for traces:

Soundness Whether every step in the trace is in the ground-truth trace. Individual steps of a trace could be unsound in several ways: (a) miscopying either the body of a function or a part of the previous line that was not being expanded in this step (we called these *transcription errors*); (b) simple arithmetic mistakes when reducing expressions; or, (c) expanding function calls within recursive parts of a data structure, rather than substituting the function body.

Completeness Whether the student listed all the steps in the ground-truth trace.

Correctness Whether the student determined the correct final result of evaluating the program. (Relevant after the first quiz.) Note that this is in principle subsumed by Soundness, but is still useful to call out explicitly.

In addition, we also recorded:

Accumulated Context Whether the student copied the accumulated context (section 3) from line to line. When students omitted context in a step, we recorded whether the omitted part was simple arithmetic or a function call.

Order of Evaluation Whether the student expanded expressions "depth-first" (the left-most and inner-most expression first), or "breadth-first" (expanding multiple expressions from the same level at the same time). If this order was not clear from the answer (such as when students use arrows to show how multiple parts of an expression expand), we coded the order as "unknown".

Granularity of Tracing Steps Whether students explicitly included steps to evaluate arithmetic expressions or just used the results of such expressions in the next step.

The authors coded the responses together, so we do not report on inter-coder reliability.

Per-Quiz Results. Figure 4 shows students' average correctness scores (ignoring soundness and completeness) across all tracing problems on each of the four quizzes. The dip in performance in quiz 3 is due to the number of students who ran out of time. The students who did not do well in quiz 4 generally suffered from transcription errors. Besides these two issues, students generally did well. The two students who did not get any questions right on the fourth quiz were two of the weaker students: they also struggled with issues outside of the substitution model. We explain specific issues encountered in each quiz in the following sections.

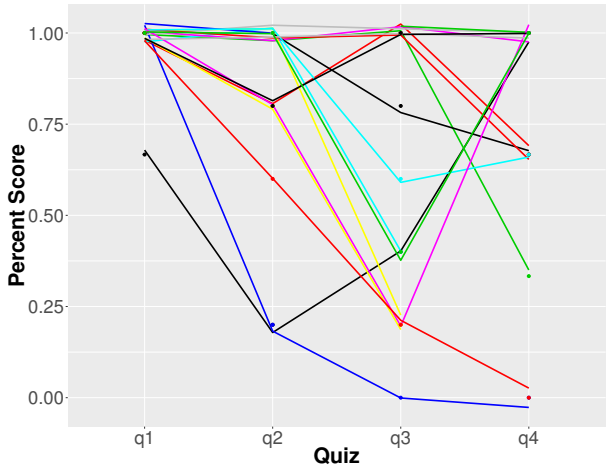


Figure 4: Student correctness over the four quizzes (lines are jittered to make individual student performance clearer)

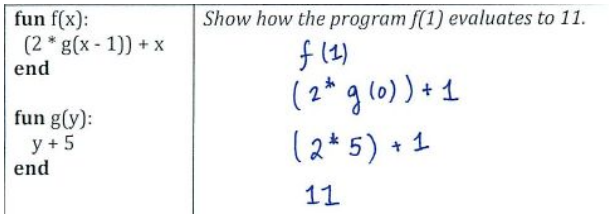
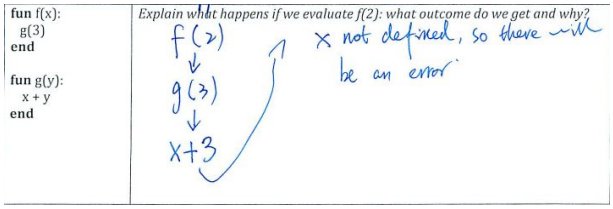


Figure 5: Tracing a program with multiple function calls

Quiz 1: Function Calls. Students were generally correct, sound, and complete across these simple substitution programs. Twelve of the 18 students who took this quiz achieved perfect scores on these metrics across the questions, with the exception of the question on dynamic scope. The following figure shows the dynamic scope question and a correct trace of it:



Nine students correctly determined that this program would yield an error, although two thought the error was that function g was undefined (despite not making this mistake on other questions). The other 9 incorrectly used the previous binding of x.

Shortcuts. Figure 5 shows a trace involving multiple function calls. This student skipped some arithmetic steps, including replacing the call `g(0)` directly with its result (without explicitly substituting into the body of `g`). Five students omitted copying simple arithmetic when evaluating `g(0)`. Some students put the evaluation of `g(0)` elsewhere on the page, inserting the result into the context when finished (akin to a simulating execution model [8]).

Misconceptions. Several students failed to substitute arguments for all instances of a function parameter: for example, four students turned `f(1)` into `2 * g(1 - 1) + x` on the problem traced previously, failing to substitute the last `x` (though they substituted later in the trace). This seems related to missing the dynamic scope question, although one of these four students answered that question correctly. On the dynamic scope question, one student thought that the value bound to `x` was stored for later access, even though this is not how the substitution model works (the course had taught that value of constants were stored for later retrieval; this student appeared to conflate this with storing parameter values). Three students substituted the expanded expression for the body of `g` (`x + 3`) back into `f`, thus accidentally changing the scope of `x`. The dynamic scope question thus proved particularly useful for uncovering misunderstandings about the substitution notional machine.

One student thought that the call `g(0)` in fig. 5 would return `0` (its argument), a pattern we see repeated on later quizzes as well.

Quiz 2: Reusing both Functions and Parameter Names. All but 1–2 students had sound and correct traces for these questions, suggesting that parameter-name reuse does not cause problems within the substitution model. Errors on these included two students who (incorrectly) reduced `h(5) + h(4)` to `h(9)`. Expressions such as `h(5) + h(4)` let us see whether students expand breadth-first or depth-first: 8 students chose the former (a form of shortcut in the substitution model).

This quiz also included the Collatz function, which was interesting because the course had not yet shown or discussed recursive calls. Thirteen (of the 18) students answered the question correctly and had sound traces. In the in-class review of the quiz after students turned it in, several asked about the new recursive pattern, though they had traced it correctly—suggesting that they noticed this new feature but, armed with substitution, were not put off by it. Among the students who did not answer correctly, three stopped evaluating on the recursive call and two simply returned the argument to the function as its result (a misconception we also saw on quiz 1, arising here with different students).

Misconceptions. The main misconceptions arose from students applying incorrect algebraic manipulations to code expressions. Two students reduced `h(5) + h(4)` to `h(9)`. Another student produced the following trace for `f(3)` (where `f` is shown in fig. 3(b)):

```
1 | x + g(x * 2)
2 | x + (2x - 3)
3 | 3x - 3
```

Rather than substitute `f`'s `x`, the student replaces the body of `g` in the body of `f`, then uses algebra—which produces the same result, but is not at all how actual evaluation works—to result in `3x - 3`.[2]

Quiz 3: Linear Structural Recursion. Most students traced recursive functions over lists correctly, for both tail- and non-tail-recursive functions. Seventeen (of 18) students correctly traced the list recursion in fig. 3(d). Ten answered the tail-recursion question (not shown) correctly; however, seven students did not finish all of the questions. We have reason to believe this is due to lack of time rather than not knowing the subject. Errors were mainly in transcription (e.g., copying the wrong variable into the next step).

Shortcuts. The introduction of lists, which are more complex data than numbers (and thus take more time to write down), inspired new shortcuts. One student mentally tracked the list bound to a variable separately rather than copy its value into the body of the function. Some students changed the written notation for a list to drop the `list:` tag required in the language; others dropped list notation entirely for single-element lists. This suggests (as we might expect) that the substitution model starts to get cumbersome with structured data.

Two students simply skipped steps that wouldn't impact the result (for a function that added zero in some cases). One student evaluated the context at the same time as evaluating the function call. It is hard to tell whether the student is using breadth-first evaluation, or whether they are simply reducing the amount of text they copy from line to line. Two students skipped substitution for the base case (the empty list as input) when it did something simple like return 0 or 1. This is similar to shortcuts seen on earlier quizzes for functions that performed simple arithmetic computations.

Misconceptions. One student misunderstood list destructuring, thinking it provided the first and *last* elements of a list, rather than the first and *rest*. Another student also had trouble with list structures, wrapping intermediate function results in lists.

Quiz 4: Tree Recursion. Most students used breadth-first evaluation on these programs. Eleven (of 15) students answered the Fibonnaci-like question correctly; three of the others had transcription errors, while one only expanded the left recursive branch.

In contrast, while 10 (of 15) students predicted the correct answer for the problem in fig. 3(e), only one produced a sound and complete trace. Most of the others suffered transcription errors (unsound) or relied on intuition about what the function would compute on subtrees (incomplete). Many who answered incorrectly dropped a "`+ 1`" term from the context, resulting in the wrong answer.

Shortcuts. Shortcuts were similar to those on quiz 3; however, three more students skipped the base case function-call on the tree-recursion question. One student who skipped the base cases in quiz 3 did not skip the base cases in quiz 4. Some questions used longer function names, which students truncated to be shorter while writing out traces.

6 SUPPORTING MUTATION

After the course switched to Python 3, it covered variables, `for`- and `while`-loops, assignment statements, and mutation of fields. As we discussed in section 3, we cannot use substitution with mutation. In the case of mutable *variables* (as opposed to fields), we can use a simpler mutable notional machine, where the names are mapped to values by mutable dictionaries. (With mutable fields we must also record the structure of the heap.) We refer to this as the *mutable environment* notional machine.

In the Python segment, the instructor taught students this new notional machine, then asked students (in the fifth quiz) to trace two programs: one a loop through a list of words that concatenated

[2]Note that terms like `3x`, in place of `3 * x`, are not even syntactically valid.

# student	loop/list	recursion/tree
7	correct	(mostly) correct
2	(mostly) correct	initial call only
4	(partly) correct	no answer
2	initial call only	initial call only
3	mix environment, evaluation, substitution, and some prose	same as loop
1	no answer	no answer

Table 1: Environment evolution for loop and tree-based recursion in Python

those words longer than 3 characters, and the other a recursive tree-traversal program.

Directly comparing these results to those on substitution is complicated because this was the first task where students had used the new notional machine. Nonetheless, we note that students performed less well on recursion with this notional machine than with substitution. More usefully, observations from this quiz provide useful feedback to affect future versions of these studies.

Table 1 summarizes students' performance on the two tracing problems with mutable environments. For the students who did not get both problems correct (all but the first row), the tree-recursion problem caused more difficulty than the loop-based one. The 6 students across the second and third rows showed a basic understanding of the mutable environment model on the list problem, but couldn't apply it to the tree-recursion problem. Three students (penultimate row) tried to integrate showing how the program evaluated (à la the first four quizzes) with showing the environment evolution asked for here. We see elements of substitution in each of these students' answers, rather than an effective switch to the new notional machine.

7 CONTRIBUTIONS, RESULTS, DISCUSSION

Our study shows that substitution is a very promising model for understanding recursion. Students were able to handle a variety of problems—including tree recursion, which is rarely considered in prior research—without much difficulty. Perhaps even more surprisingly, with substitution they were able to trace non-trivial problems like the Collatz function even before being introduced to recursion. While these studies have to be reproduced on different student populations, this does suggest that the difficulty at least of *tracing* recursion needs to be reconsidered.

As we have noted, substitution does not easily work with mutable state. Most curricula have taken state for granted, and hence cannot consider this notional machine. However, many curricula in widespread use (such as HtDP and Bootstrap [bootstrapworld.org]) are "functional first", and can and do use substitution. We even have preliminary evidence (section 6) that an imperative notional machine may not be easy. Thus, we need a broader discussion about the style of programming used in introductory curricula.

More subtly, our observations highlight an interesting difference between substitution and mutable environments: substitution requires only one notation (a program in the syntax of the language) to capture execution and the current state of the evaluation. The mutable environment notional machine not only has more notation, it also fails to directly capture return values from functions and must be expanded to handle them. Students must then learn to navigate all these components when tracing programs.

We have also identified weaknesses with the substitution model. It can require too much copying of context, causing students to take shortcuts, which can in turn get them into trouble. It also becomes unwieldy when dealing with large data structures, further inducing shortcuts. This suggests a need for a better model that enjoys the benefits of substitution while avoiding these problems.

We have found the use of programs with unbound variables useful at unearthing student misconceptions and hence evaluation models. This fits the general pattern of giving students erroneous programs to better probe their understanding. However, we are not aware of prior recursion research using this particular idea.

We believe there are also several concrete takeaways for instructors. First, consider the use of substitution (in a mutation-free setting) for teaching function application leading up to recursion. Second, use erroneous programs—e.g., with unbound identifiers—to probe student understanding. Finally, consider using our learning progression for teaching recursion.

ACKNOWLEDGMENTS

We thank Natasha Danas, Matthias Felleisen, and our reviewers. We also thank Colleen Lewis for answering questions on the design of her study [8]. This work is partially supported by the US National Science Foundation. First author's last name is "Tunnell Wilson" (index under "T").

REFERENCES

[1] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an algebraic stepper. In *European Symposium on Programming*.
[2] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* (1986), 57–73.
[3] Matthias Felleisen. 1987. *The Calculi of Lambda-nu-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. Ph.D. Dissertation. Bloomington, IN, USA.
[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press.
[5] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *ACM Symposium on Computer Science Education*. ACM, New York, NY, USA.
[6] Hank Kahney. 1983. What Do Novice Programmers Know About Recursion. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 235–239.
[7] D Midian Kurland and Roy D Pea. 1985. Children's mental models of recursive LOGO programs. *Journal of Educational Computing Research* (1985), 235–243.
[8] Colleen M. Lewis. 2014. Exploring Variation in Students' Correct Traces of Linear Recursion. In *ACM International Conference on Computing Education Research*. ACM, New York, NY, USA, 67–74.
[9] Renee McCauley, Brian Hanks, Sue Fitzgerald, and Laurie Murphy. 2015. Recursion vs. iteration: An empirical study of comprehension revisited. In *ACM Symposium on Computer Science Education*. ACM, 350–355.
[10] Greg L. Nelson, Benjamin Xie, and Andrew J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *ACM International Conference on Computing Education Research*. 2–11.
[11] Ian Sanders, Vashti Galpin, and Tina Götschi. 2006. Mental models of recursion revisited. In *ACM SIGCSE Bulletin*. ACM, 138–142.
[12] Amber Settle. 2014. What's motivation got to do with it? A survey of recursion in the computing education literature. *Technical Reports* (2014).
[13] Joe Tessler, Bradley Beth, and Calvin Lin. 2013. Using cargo-bot to provide contextualized learning of recursion. In *ACM International Conference on Computing Education Research*. ACM, 161–168.
[14] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can we Crowdsource Language Design?. In *Onward!'17*. ACM, New York, NY, USA.
[15] Susan Wiedenbeck. 1989. Learning iteration and recursion from examples. *International Journal of Man-Machine Studies* (1989), 1–22.